SDF Workshops

Futures

Why Futures ?

JVMs threads are implemented using native threads.

Costly:

- Context switch time
- Memory consumption

Scala Futures provide light weight concurrent tasks running within
a small number of threads (execution contexts).

What is a Future ?

A `Future[A]` eventually (or immediately) produces a value of type `A` or a `Throwable`.

Creating a Future

`Future.apply` takes an expression (a by-name parameter) and schedules it for evaluation in the thread pool of an `ExecutionContext` (passed implicitly).

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

Future(1 to 10000000 sum)
```

Creating a Future without scheduling a task

```
import scala.concurrent.Promise
import scala.util.Success

val promise = Promise[Int]()
promise.future
promise.complete(Success(10))
```

Sequencing Futures

The future of the future is just the future, so we can flatten:

`Future[Future[A]] => Future[A]`

and we can sequence instances with `flatMap`:

```
val evenFuture = Future(2 to 10000000 by 2 sum)
Future(1 to 10000000 by 2 sum)
   .flatMap(odd => evenFuture.map(_ * odd))
```

For concurrency, `evenFuture` must not be created within the call to `flatMap`.

Sequencing a collection of Futures

```
val futureSeq = Future.sequence(
  (1 to 10000).map(n => Future(n * n)))
//futureSeq: Future[IndexedSeq[Int]] = Promise$DefaultPromise@4a9829da
```

To have this thread await the result of the computation we must use `Await.result`

```
import scala.concurrent.Await
import scala.concurrent.duration._

Await.result(futureSeq, Duration.Inf)
//res8: IndexedSeq[Int] = Vector(1, 4, 9, 16, 25, ...)
```